Developer's Guide for Libra 1.1.1

Daniel Lowd <lowd@cs.uoregon.edu>
Amirmohammad Rooshenas <pedram@cs.uoregon.edu>

May 21, 2015

1 Introduction

Libra's code was written to be efficient, concise, and have few external dependencies. This document describes the design of the Libra toolkit, and how to use or extend its functionality.

2 Implementation

Libra is implemented in OCaml, a programming language in the ML family that supports procedural, functional, and object-oriented programming paradigms. Libra was developed using OCaml 4.02, and may not be fully compatible with earlier versions.

2.1 Why OCaml?

Programs written in OCaml can be compiled to either byte code or native code. Native code performance is much faster than Python and often competitive with C++. This speed is important when implementing algorithms that could run for hours or days. OCaml is statically typed with automatic type inference, which leads to programs that are much more concise and easier to refactor than similar programs written in Java and C++, and safer than programs written in Python. OCaml has a good foreign function interface for connecting to C libraries, such as expat, lbfgs, and our own optimized pseudo-likelihood computation code.

As an experiment, we compared the speed of the Libra Gibbs sampler to a similar one that we wrote in C++ some years ago. To our surprise, the OCaml implementation was actually 23% faster! Theoretically, a carefully optimized C++ program should almost never be slower than one in OCaml. In practice, however, working in OCaml can make it easier to optimize code, leading to a faster implementation given limited development time.

2.2 Programming Style

Libra uses both procedural and functional paradigms. Functional paradigms can lead to faster development and more concise code, but procedural methods are often necessary for obtaining faster code that uses less memory, making some algorithms easier to implement. The overriding goal in Libra is practicality, not purity. Hence, mutation is used fairly heavily.

We have chosen not to use the object-oriented features of OCaml. (OCaml's variant types are typically the preferred approach when inheritance is not required.) Named and optional parameters are an OCaml feature that we are not currently using, but may use at some point in the future.

Libra does not depend on any external libraries, except those included in the distribution.

3 Organization

All source code is in subdirectories of the `src/` directory. Each subdirectory is devoted to either a library (e.g., **data**, **bn**, etc.) or one or more programs (e.g., **acbn**, **util**, **inference**, etc.).

A hierarchy of Makefiles follows the directory hierarchy. The master Makefile for source code is `src/Makefile`, which calls the Makefiles of the subdirectories. Shared definitions are placed in `src/Makefile.shared`, which is included by the sub-Makefiles. Much of the work is done by OCamlMakefile (included), written by Markus Mottl, which automatically detects dependencies and performs compilation and linking. After compilation, all libraries and their interfaces are placed in `lib/` and all binaries are placed in `bin/`.

To rebuild all libraries and programs, run:

```
make clean; make
```

4 Supporting Libraries

The toolkit employs nine libraries: **ext**, extensions to the standard library; **data**, reading and writing of examples; **mn**, Markov networks and log-linear models; **bn**, Bayesian networks and dependency networks; **circuit**, arithmetic circuits; **spn**, sum-product networks; **pll**, optimized computation of pseudo-likelihood and its gradient; **lbfgs**, the limited-memory BFGS optimization algorithm; and **expat**, the Expat parser. The last two libraries consist of OCaml bindings to the original libraries written in C. The L-BFGS code was originally written in FORTRAN by Jorge Nocedal and ported to C by Naoaki Okazaki. Expat was written by James Clarke, and the OCaml bindings were developed by Maas-Maarten Zeeman.

We describe the libraries that were developed internally to Libra in more detail below. For full documentation of each library's interface, see the automatically generated API reference available in the `doc/html` subdirectory of Libra.

4.1 Ext Library

The **ext** library includes miscellaneous utility functions, extensions to the OCaml standard library (including List and Array), and two utility modules, Timer and Heap (priority queue implemented as a binary heap). If you wish to add general-purpose functions, such as an `Array.map4` function that performs a `map` over four lists at once, **ext** is the place to implement it. **ext** also includes common arguments, shared mechanisms for working with log files, probability distributions, and more.

4.2 Data Library

The **data** library includes facilities for reading discrete-valued data points from a file, writing them to output streams, reading in variable schemas (represented as an array of variable dimensions), checking that a data point is valid according to a given schema (i.e., having the correct length and a legal value in each dimension), and reading and writing lists of marginal distributions. Future functionality for this library could include support for other file formats, such as ARFF, or more general-purpose manipulation methods.

4.3 Mn Library

The **mn** library reads, writes, and represents Markov networks with factors represented as tables, trees, sets of features, or individual features. A dummy "constant" factor is also allowed. Most of the content is in `factor.ml`, which defines the different types of factors, how to get the factor value for different configurations, how to convert among different types of factors, how to simplify them given evidence, and how to write them to a file or stream.

File I/O for the `.mn` format is handled by `mnLexer.mll` and `mnParser.mly`. The UAI file format is handled by `uaiFormat.ml`. See these files for examples if you wish to add more file formats.

Some preliminary framework is also present for tied weights, but it is not currently being used.

4.4 Bn Library

The **bn** library reads, writes, and represents Bayesian networks (BNs) and dependency networks (DNs) with conditional probability distributions (CPDs) that are trees, tables, or arbitrary sets of factors. The basic type is defined in `bnType.ml`, which is then included by `bn.ml`. Files `bifFile.ml`, `xmodFile.ml`, and `cnFile.ml` support reading and writing Bayesian networks in the Bayesian interchange format

(BIF), WinMine's .xmod format, and Libra's .bn format, respectively. The reason for the split between `bnType.ml` and `bn.ml` is to avoid circular dependencies, since the file formats depend on the Bayesian network data type, and we wish to make these file formats available through the single interface of `bn.ml`.

Methods for accessing a BN include computing the conditional probability of a node given a state vector that specifies the values of its parents, computing the probability of a node given its Markov blanket, converting to a Markov network, generating a sample from an empty network (i.e., one with no evidence), and more. Methods for setting CPDs are provided, but are somewhat limited.

DNs are differentiated by the `acyclic` flag, which is false for DNs and true for BNs. This mainly affects how Markov blanket probabilities are computed, which only depend on the parents in a DN.

Future functionality could include better interfaces for manipulating the structure and parameters of a BN, support for other file formats (such as the XML-based BIF format), and better error handling.

4.5 Circuit Library

The **circuit** library reads, writes, and represents arithmetic circuits. An arithmetic circuit is a directed, acyclic graph in which each interior node is a sum or product of its children, and each leaf is an indicator variable or numeric constant. Arithmetic circuits can be used to represent any log linear model. Arbitrary marginal and conditional probabilities can be computed in linear time in the size of the circuit, but the circuit may have exponential size in the original model. See Darwiche (2003) for more information on arithmetic circuits. Arithmetic circuits are very closely related to sum-product networks described by Poon and Domingos (2011); see Rooshenas and Lowd (2014) for more details on the relationship between the two representations.

The file `node.ml` defines a data type representing a node in an arithmetic circuit, which is one of the following types:

- `TimesNode`: the product of its children
- `PlusNode`: the sum of its children
- `VarNode`(var, value): the indicator variable which is 1 if the specified variable can take on a particular value and 0 otherwise
- `ConstNode`(w): a constant with value e^w
- `DummyNode`: a placeholder for representing a "null" node

It includes basic facilities for constructing, evaluating, and even iterating over a graph of nodes. It also defines sets and hash maps of nodes.

All node definitions are included by `circuit.ml`, which defines the circuit data type and methods for constructing circuits and using them for inference.

4.6 Spn Library

The **spn** library reads, writes, and represents sum-product networks (SPNs). It has also been used for representing mixture of trees. Each node in an SPN defines a distribution as one of the following:

- `TimesNode`: the product of the distributions represented by its children.
- `PluseNode`: the mixture of the distributions represented by its children.
- `LeafNode`: a uni-variate or multi-variate distribution.

The SPN library provides a set of methods for creating SPN structures, as well as horizontal (sample) and vertical (variable) partitioning methods for learning SPN structures. These partitioning methods have been used in Gens and Domingos (2013) and Rooshenas and Lowd (2014).

The definition and methods are included in `spn.ml`.

4.7 Pll Library

The **pll** library performs efficient computation of pseudo-likelihood and its gradient, given a Markov network represented as a set of weighted conjunctive features. It includes implementations in both OCaml and C. The C implementations are recommended, since they have been more heavily optimized. See the code for `mscore` and `mnlearnw` for example usage.

5 Modifying Libraries

If you wish to add an algorithm to Libra and you need additional functions in one of the libraries, you are encouraged to add this functionality. If you change a library's implementation, you will need to recompile all programs that depend on the library yourself using commands such as the following.

```
cd src
make clean; make
```

If you modify a library's interface as well, then you must modify the interface accordingly. You can use the available automated testing, to make sure that the new changes do not break some other parts of the toolkit.

6 Automated Testing

A few automated tests are available in the `test/` directory. To run all of them, use the `runall.sh` script:

```
cd test
./runall.sh
```

Some of these tests are sensitive to differences in the random number generator or floating point arithmetic. These "fragile" tests are run at the end of the `runall.sh` script. Failure of a fragile test does not necessarily indicate broken code, just a difference of some kind. When one or more tests fail, you can compare the generated output files to the test files.

Each subdirectory of `test/` contains tests pertaining to the programs from the corresponding subdirectory of `src/`. For instance, `test/util` contains tests for `mscore` and `bnsample`. Tests within a single directory are run using the `run.sh` script. For example, to run just the `util` tests, execute the following:

```
cd test/util
./run.sh
```

Fragile tests are run by the `run-fragile.sh` script, if present.

Note that these are not unit tests. They test the complete functionality of programs, not the individual functions within them. Their main use at this point is to make it easier to identify when a modification has resulted in broken functionality.